CONSIDERING THE DETAILS

Bob Perrin

Pedigree Protection—Watchdog Circuits

A motion-triggered recording of vicious canines may keep the average cat burglar off of your property, but it takes more than just the presence of a watchdog circuit to keep embedded-system failures at bay. Sit, stay, and listen while Bob explains how effective watchdog timers are implemented in hardware and software.

A favorite axiom of one of Z-World's engineers is, "Once a year, all engineers should be put on a life support system they helped design." Many engineers have come to regard electrical engineering as nothing more than playing with techno-Legos. Need gain? Grab an op-amp. Want a sensor's output in digital form? Grab an ADC. Want to crunch some numbers? Grab a PIC. Want to drive a pump? Grab a big FET.

Engineering is a delicate blend of art and science. Silicon manufacturers are rolling big mojo into tiny chips, but developing reliable hardware still requires relentless attention to detail. Designing robust systems will never be as simple as plugging together Legos. Considering The Details is devoted to examining the details associated with practical engineering.

APPLYING WATCHDOGS TO EMBEDDED SYSTEMS

Watchdog circuits have been around for decades. Watchdogs are the first things people bring up when they discuss the design of "robust embedded controllers." Just the presence of a watchdog in a system gives the designer the sense that no matter what goes wrong, the system will reset and be back up in seconds. Unless the software is written carefully, this is a false sense of security.

FAILURE IN EMBEDDED SYSTEMS

The nature of an embedded system is one of stealth and reliability. A well-designed embedded controller is innocuous to the end user. For example, a pilot doesn't care that his plane carries more computing horsepower than was used to send the Apollo mission to the moon. He only cares that the plane turns left when he tells it to.

Users of these systems may not have a clue about electronic design or how code works. They only know about their end application and how the instrument they paid money for is supposed to help them accomplish their task.

To be successful, embedded systems must be designed to withstand adverse treatment and be tolerant of abuse by ignorant users. For example, a user may mistakenly (or purposely) tell the system to perform a task that is destructive to the system. Well-designed systems will be able to recover from failures and continue to function.

Embedded systems may be subject to all sorts of EMI, ESD, RFI, power spikes, brownouts, shorts on I/O pins, and general abuse. This can lead to unexpected failure modes in both hardware and software.

Failure modes can range from a simple corruption of a bus transfer to corruption of internal CPU registers. I/O devices may latchup. Baud-rate generators may have reload values corrupted. Battery-backed nonvolatile RAM may become corrupt. File systems may become corrupt. Opened files may not get closed properly because of spurious system resets.

The exact effects of these failures on a system are difficult to predict. For example, assume a single bit is corrupted in the CPU's status register by EMI generated from a nearby ESD event. Perhaps a subsequent test of that bit will cause an errant branch in the kernel or application. However, if the bit is not used or is reinitialized by subsequent instructions, there will be no apparent problem.

An equally likely scenario is that of a single corrupt bit in the program counter. If the bit is a high-order bit and the system's physical memory is small enough to not decode the high-order bits so the logical memory just repeats itself right through the CPU's address space, nothing bad will happen. More likely, the CPU will begin to fetch instructions out of the sequence intended and the application framework will become unrecoverably corrupt. When an application framework becomes corrupt or a CPU latches up, the last line of defense is a hardware watchdog. But even a dead-simple hardware watchdog is not a panacea of reliability unless careful attention is paid to the design of the firmware interface to the watchdog.

WHAT IS A WATCHDOG TIMER?

A watchdog circuit in its simplest form is just a counter. When an overflow happens, the watchdog asserts the system RESET line and the controller resets. Under normal operation, the code running in the controller will periodically clear the watchdog counter (hit the watchdog) so that it will not overflow.

When the controller enters an infinite loop, or gets lost executing an unintended sequence of instructions, the controller will fail to hit the watchdog. After a while, the watchdog will overflow and reset the controller. And the controller will shortly be back on duty, running properly.

The trick to making this scheme work is writing code that will only hit the watchdog under controlled conditions.

HOW ARE WATCHDOG TIMERS IMPLEMENTED?

Watchdog timers can be implemented with discrete components. Figure 1 shows a simple circuit that uses an LMC555 timer as an oscillator and a 74HC161 as a counter. This circuit works fine but is pretty expensive and requires a lot of board space.

Because we don't want to see the watchdog held in reset by a stuck I/O line, the watchdog reset line is designed to be sensitive only to transitions. That's what the circuits around Q1 and Q2 do.

A more integrated implementation is the dedicated supervisor chip. These chips come in many flavors from many manufacturers. They all pretty much have varying mixes of the following features:

- power monitoring
- reset generation
- watchdog
- backup-battery switchover
- SRAM write protection

Some supervisor chips, like the Analog Devices ADM691, are second sourceable (Linear Technology's LTC691 and Maxim MAX691). Most are not.

Some parts offer enhancements to the basic watchdog function. For example, the Analog Devices ADM696 has two different timeout periods. The first timeout period after a system reset is long. After the processor hits the watchdog, the ADM696 switches to a short timeout period. The difference between the long and short timeout periods is about a factor of three.

The engineers at Analog Devices were thinking overtime on this one. When the system first powers up and RESET deasserts, the code may take a bit of time to initialize variables, frames, and stacks. The initial long timeout on the watchdog gives the processor time to finish all of its initialization and start the application.

After the application is up and running, it will periodically hit the watchdog. After this occurs, the watchdog assumes the code is running correctly and switches to a short timeout period.

Commercially available supervisors often enable you to set the length of the timeout by using external passive components. The timeout may also be a function of the supply voltage.

The final common method for implementing watchdogs is to select the system's microcontroller or microprocessor that already has an integrated watchdog on the same dice.

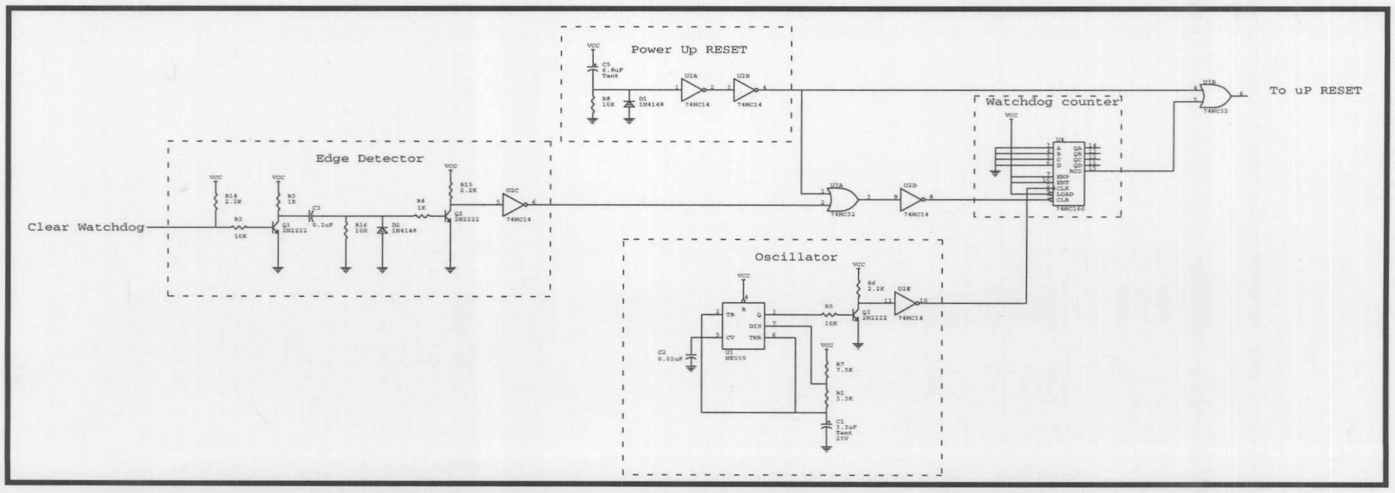

Figure 1—*Back in the '80s, watchdog circuits were simple but required more than a few parts.*

HANDLING WATCHDOGS IN SOFTWARE

The trick to making a watchdog perform properly is software. If the code gets stuck in an infinite loop and something in that loop hits the watchdog, the watchdog will fail to reset the processor and the system will hang. The watchdog will be effective as long as the code is written to minimize or eliminate the likelihood of this undesirable condition.

There are two models for how software interacts with hardware. The first model is the simplest. Application-level software talks directly to the hardware. The second method places a layer of abstraction between the hardware and the application. This layer is called a driver. The application talks to the driver and the driver talks to the hardware.

A commonly touted principle says, "for reliability, systems should be made as simple as possible." At first glance, this appears to imply that the application should talk to the watchdog hardware directly. However, on closer examination, we will see that this is exactly the wrong approach for system reliability.

For the sake of our discussion, `hitwd ()` will be the simplest function possible to hit the watchdog. You can think of `hitwd ()` as a macro that expands into some inline assembly that resets the system's hardware watchdog.

Typically, applications that talk to the hardware watchdog directly have calls to `hitwd ()` sprinkled throughout the code. It's easy for programmers to lose sight of the watchdog's intended function and only be concerned about hitting the watchdog often enough to keep the system from resetting.

After all, software engineers usually have their hands full contending with the application development and debugging. The software may have to implement complex tasks involving control of mechanics, logging data, doing ugly computations quickly, and processing acquired data. Under these conditions, the easiest way for the programmer to deal with the watchdog is to cram a call to `hitwd ()` into every 20 lines of code.

When code evolves like this, the watchdog becomes almost superfluous. Many issues that aren't normally encountered in desktop systems can cause embedded systems to fail. The contents of CPU registers, RAM, EEPROM, flash memory, and EPROM can become unpredictably corrupt when the system is subjected to EMI, ESD, or power upsets.

These events don't usually cause the CPU to stop executing code. Most often the CPU continues to execute the code, but with out-of-sequence or unintended instructions. Another common failure mode is for the CPU to execute properly sequenced instructions but operate on corrupt data, say from an upset PIO. If the application is calling `hitwd ()` frequently, the watchdog reset may never trip.

Harvard architectures are potentially more prone to executing instructions out of sequence. For example, consider the ubiquitous 8051. Only 64 KB of code space is available. If the application is large, then almost all of the code space will be filled with legitimate instructions. If the PC becomes corrupt, the CPU will execute instructions that are not in the desired sequence. However, because the CPU can only fetch instructions from code space, the instructions fetched will be part of the application. And if calls to `hitwd ()` are liberally sprinkled in the code, then the watchdog will continue to be reset, even though the application has failed.

In a Von Neumann machine, if the PC becomes corrupt, instruction fetches may occur from the application or from data. In this case, it is somewhat more likely that the watchdog will trip, as the possibility of data containing the proper sequence of "instructions" (data) to cause the watchdog to be hit is probably small. However, this completely depends on the contents of the data and the instruction set of the processor.

Now let's examine what type of drivers we can interpose between the application and the watchdog hardware. I'm familiar with two types. One is an interrupt-based system; the other involves a called driver. Both schemes check the integrity of the system's application framework before hitting the watchdog.

People often consider it a bad idea to hit a watchdog in an ISR. In this situation, the main application may fail, but a periodic interrupt may still cause the ISR to reset the watchdog. Thus the embedded system crashes, but the watchdog doesn't reset the system as long as the periodic interrupt is still functioning.

This situation only occurs if the ISR doesn't perform an adequate sanity check on the application(s), which can be accomplished by setting up virtual or software watchdogs. A virtual watchdog is simply a software counter that the application increments and the ISR decrements. If the virtual watchdog reaches zero, the ISR determines that the main application is not functioning properly, then the ISR disables interrupts and enters an infinite loop to wait for the hardware watchdog to reset the system.

A single virtual watchdog is no more effective than simply scattering `hitwd ()` calls throughout the code. However if an array of virtual timers is set up, the ISR then has to check on multiple points in the main application.

If multiple tasks are running concurrently, an array of virtual watchdogs can be used to verify the sanity of each task. If any task fails, the ISR can determine if the hardware watchdog should be tripped or if the kernel should be notified of the failed task so that the kernel can deal with the errant task.

An array of virtual watchdogs is much less likely to be properly serviced by a corrupt application framework than code with multiple direct accesses to the watchdog hardware. Therefore, the overall system reliability is enhanced.

An addition to software watchdogs, the ISR can perform a sanity check on the stack pointer(s). One common symptom of a rampaging CPU is a corrupt or overflowed stack. It's a good practice to bound-check the stack pointers before hitting the hardware watchdog in an ISR. Depending on the specific application, other system checks may be made by the ISR—for example, checking the status of critical I/O or doing a CRC in data in SRAM.

There are two ways an errant CPU can execute instructions from the ISR. One is for the errant CPU to execute instructions while linearly approaching the ISR code. The second is for the errant CPU to branch the execution path into the middle of the ISR code. There are two safeguards that should be used to prevent this from happening.

First, the instructions immediately before the ISR should be an infinite loop. If a rampaging CPU linearly approaches the ISR, the infinite loop will execute and the CPU will hang. Eventually, the ISR will see that the virtual watchdogs are not being serviced, and the hardware watchdog will reset the system.

The second safeguard mitigates the problem of a rampaging CPU jumping into the middle of the ISR and incorrectly hitting the watchdog. To complete this safeguard, a mechanism must be in place for the ISR to do a sanity check on itself.

The first instructions in the ISR should fill a fixed memory location with a sentinel value. Just before the ISR hits the watchdog, the memory location is checked for the proper sentinel value. If this sentinel value is not found, the watchdog is not hit. The last thing the ISR should do is clear the ISR's memory location, lest the sentinel value remain.

If C is used to write this type of code, a static variable can be used to allocate space for the memory location. Most compilers will initialize static variables the first time through execution. Be careful not to use a sentinel value that is the same as the compiler's initialization value. Otherwise, the sanity check will not be as bulletproof as it should be.

The idea is to only hit the watchdog in a single place in the code and then jealously guard that small piece of code. If the application doesn't jump through all the right hoops or if the stack, data structures, or I/O appears to be corrupt, the watchdog is not hit.

The ISR/virtual-watchdog scheme requires careful design. As the number of systems checks and virtual watchdogs grows, so does the possibility of properly functioning, but ill-conceived code that fails to behave in accordance with the scheme and causes a system reset.

These are the issues and tradeoffs the engineer must weigh. Designing high-reliability code for harsh environments or embedded controllers is not a trivial task. But that's why we get paid the big bucks. Attention to detail is the stock and trade of engineers.

In addition to the ISR/virtual-watchdog scheme, there is a simpler called-driver scheme. It is most applicable to systems where a periodic interrupt is unavailable or only a single application is running. The application calls the driver, and the driver checks the system integrity before hitting the watchdog.

The driver should have an infinite loop preceding it, just as the ISR above. The driver can use a sentinel value to ensure the driver was entered at the start of the function. The driver can make the same type of system integrity checks as the ISR.

The called driver has one additional check that the ISR cannot do. Because the driver is called from fixed locations in the application, the driver can examine the stack and verify that the return address is valid.

This check has to be hand coded because compilers don't generate this type of information. A bit of work, yes, but depending on the system requirements, worth it. Hand-crafting assembly is often required where solder meets code.

As with any safety measure, if not used properly, a watchdog becomes a piece of wasted hardware serving no more function than to help the designer sleep at night. However, a properly implemented watchdog can save the day.

Bob Perrin spends his days designing general-purpose C-programmable embedded controllers and troubleshooting customer system-level problems for Z-World (www.zworld.com). Over the last ten years, Bob has designed instrumentation for agronomy, soil physics, and water activity research. He was also the lead design engineer for an intrinsically safe line of workstations for use in explosive gas and particulate environments (class 1, divisions 1 and 2). For more articles by Bob, visit his online library at www.engineerbob.com.

For their insights and contributions, many thanks to Norm Rogers, President, Z-World; Greg Young, Manager Technical Support Department, Z-World; Tak Auyeung, Ph.D.; and Joel Baumert, software engineer.

SOURCES

ADM691, ADM696
Analog Devices
(617) 329-4700
Fax: (617) 329-1241
www.analog.com

LTC691
Linear Technology
(408) 432-1900
Fax: (408) 434-0507
www.linear-tech.com

MAX691
Maxim Integrated Products
(408) 737-7600
Fax: (408) 737-7194
www.maxim-ic.com